Adaptive Software Engineering
G22.3033-007

Session 3 - Main Theme
Software Development Life Cycles (SDLCs)

Dr. Jean-Claude Franchitti

New York University
Computer Science Department
Courant Institute of Mathematical Sciences

Agenda

- Review of SDLC
- Environmental Diagrams
- Traditional Life Cycle Models
- Alternative Techniques
- Architectural Principles
- Use Case Driven Development
- Extreme Programming
- Agile Software Development
- Roles and Types of Standards
  - ISO 12207: Life Cycle Standard
  - IEEE Standards for Software Engineering Processes and Specifications
- Summary
  - Course Assignments
  - Course Project (Project #1 extended)
  - Readings

Part I

Review of SDLC

What is a SDLC

System Development Life Cycle:

- It is developing a computer system
- It concerns a process which takes from two months to two years
- This is called a system development life cycle

What is a SDLC

There are two forms:

- Rapid (Prototype)
  - Plan and Elaborate
  - Developmental Cycle 1
  - Developmental Cycle 2
- And Waterfall (classical)

What is a SDLC

- Waterfall (classical)
  - Requirements
  - Analysis
  - Design
  - Construction
  - Implementation

What is a SDLC

Both forms are followed by a maintenance cycle:

- Maintenance is the most expensive part
- If all the steps are done carefully maintenance is reduced
- For maintenance to be effective , documentation must exist

What is a SDLC

The system really consists of two parts:

- Model
  - Prototypes
  - Diagrams and supporting Documents
- System
  - Hardware
  - Software

Definitions

Prototype:

- A first system usually done with a rapid development tool
- Usually has limited functionality
- Users can see results very quickly

Definitions

- Planning
  - The process of gathering what is needed to solve a business problem
  - Includes a feasibility study
  - Includes project steps

Definitions

- Analysis
  - The process of determining detail requirements in the form of a model

Definitions

- Design
  - The process of drawing blueprints for a new system

Definitions

- Construction
  - The actual coding of the model into a software package
  - Uses one of three languages:
    - Java
    - Smalltalk
    - C++

Definitions

- Implementation
  - Doing whatever is necessary to startup a system
  - Includes:
    - Database
    - Networks
    - Hardware configuration

Definitions

- Maintenance
  - Doing whatever is necessary to keep a system running
  - Includes:
    - repairs to correct errors
    - enhancements to accommodate changes in requirements

Deliverables

- Deliverables consist mainly of diagrams and their supporting documentation
- For example:
  - Models that emphasize dynamics
  - Models that emphasize structure
  - Models can be used for specifying the outcome of analysis
  - Models can be used for specifying the outcome of design

Deliverables

Planning:

- System Functions
- A simple list of each requirement a system must do
- For example:
  - record video rental
  - calculate fine

Deliverables

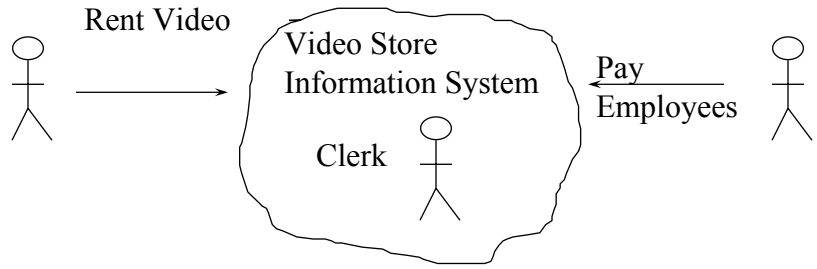
Planning:

- System Attributes
- A simple property describing each requirement of a system
- For example:
  - record video rental under 15 seconds
  - calculate fine and return response in 5 seconds

Deliverables

- Planning:

Environmental Diagram



Deliverables

Planning:

- Prototype
- Recall it is a first system usually done with a rapid development tool
- Since users can see results very quickly they will pay attention
- Final product is seldom created in same tool as the prototype

Deliverables

Analysis:

- Use case
- Shows the dynamics between the users (actors) of the system and the system itself
- This is a narrative representation

Deliverables

Analysis:

- Conceptual Diagram
- Shows the structure of the objects and their relationships
- This is a graphical representation

Deliverables

Analysis:

- System Sequence Diagram
- Shows the dynamics between the users (actors) of the system and the system itself
- This is a graphical representation

Deliverables

Analysis:

- Contracts
- Shows the state of each object before each action
- This is a narrative representation

Deliverables

Design:

- Interaction Diagram
- Shows the interaction between objects
- This is a graphic representation
- It is a dynamic blueprint

Deliverables

Design:

- Class Diagram
- Shows the structure between objects
- Shows the structure inside objects
- This is a graphic representation
- It is a static blueprint

Summary

UML provides a standard for the following artifacts:

- Use Case (Dynamic Analysis Output)
- Conceptual Model (Static Analysis Output)
- Interaction Diagram (Dynamic Design Blueprint)
- Class Diagram (Static Design Blueprint)

Part II

Traditional Life Cycle Models

Traditional Life Cycle Models

- Waterfall
- V
- Phased
- Evolutionary
- Spiral
- CBSE
- Group Exercise #1 (groups will present in class):
  - Research and Put Together a Comparative Write-up

Part III

Alternative Techniques

Alternative Techniques

- Group Exercise #2:
  - Research and Document the Main Benefits of the following techniques and how they relate to traditional life cycle models
    - RUP (Rational Unified Process)
    - RAD (Rapid Application Development)
    - JAD (Joint Application Development)
    - PSP/TSP (Personal/Team Software Process)
    - Prototyping
- Structured Application Design
- Support Technologies (e.g., MDA, Aspect-Oriented Programming, etc.)

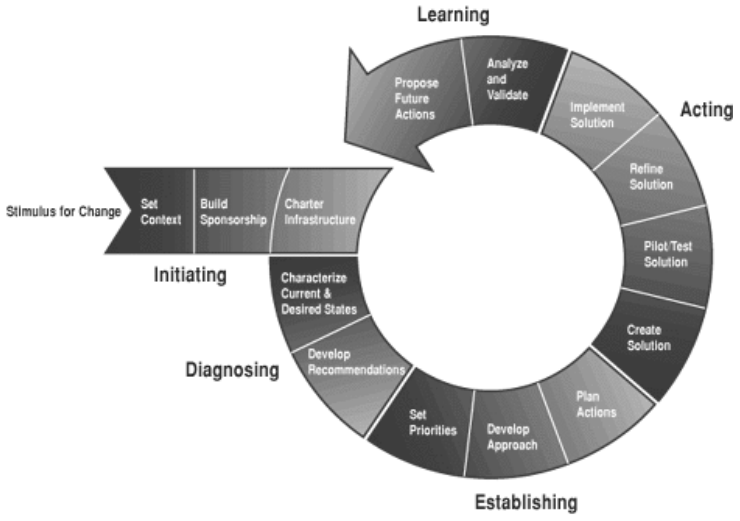
1. CMM & PSP/TSP

http://www.sei.cmu.edu

- The Capability Maturity Model for Software (SW-CMM) is used by organization to guide their software process improvement efforts
- Personal Software Process
  - http://www.sei.cmu.edu/tsp/psp.html
- The Team Software Process (TSP) was designed to implement effective, high-maturity processes for project teams
- If all projects in an organization are using the TSP, does the organization exhibit the characteristics of high process maturity, as described in the SW-CMM?
  - http://www.sei.cmu.edu/pub/documents/02.reports/pdf/02tr008.pdf

2. SEI's IDEALModel

- IDEAL is an organizational improvement model



3. Business Engineering Methodology

- Business Model/Architecture
  - Use Case View/Model
- Application Model/Architecture
  - Logical and Process View/Models
    - Content, Data, and Process Model (e.g., OIM's knowledge management, and database/datawarehousing models)
- Application Infrastructure Model/Architecture
  - Implementation View
    - Component Model (e.g., OIM's component and object model)
- Technology Model/Architecture
  - Deployment View/Model
- See Session 2 Handout on "Business and Application Architecture Engineering"

4. XML-Based Software Development

- Business Engineering Methodology
  - Language + Process + Tools
  - e.g., Rational Unified Process (RUP)
- XML Application Development Infrastructure
  - Metadata Management (e.g., XMI)
  - XML APIs (e.g., JAXP, JAXB)
  - XML Tools (e.g., XML Editors, XML Parsers)
- XML Applications:
  - Application(s) of XML
  - XML-based applications/services
    - MOM & POP
    - Other Services
  - Application Infrastructure Frameworks

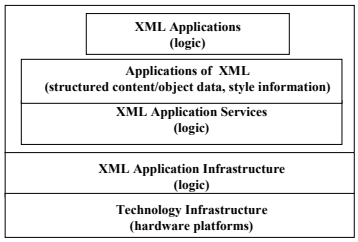
XML Metadata Management

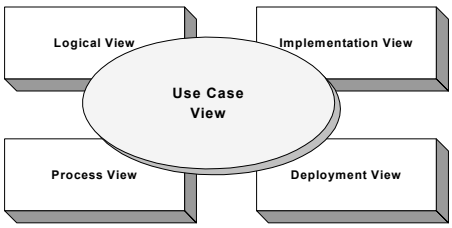
- Issue: UML may not provides enough modeling views and enough expressive power in each view to represent a complete application
- Possible Solutions:
  - Extend UML:
    - Started as the OIM Analysis and Design Model (now OMG's MDA)
  - Use Different Modeling Languages:
    - See later handout on "XML Information Modeling" (uses different models such as UML, XML, and ORM)
  - Use a Meta-Model: MOF and XMI
    - See later handouts on "UML, MOF, and XMI" and "OMG's XML Metadata Interchange Format (XMI)"
    - Design XML Schemas using UML:
      - http://www-106.ibm.com/developerworks/library/x-umlschem/

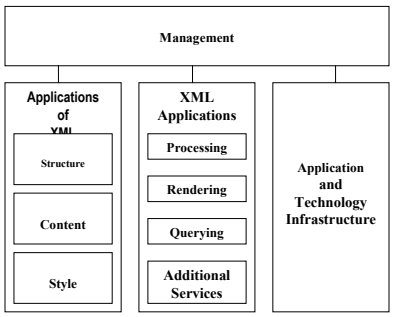
Class Project Addendum

- Project Description
  - The project will consist of providing business and application models in support of custom XML-based services handling various aspects of a chosen portable application. The actual application can be targeted to end-users (B2C), businesses (B2B), developers (toolkit). As an example, you could model an XML-based training studio supporting VoiceXML, and application-sharing capabilities.
  - Sample target applications relevant to the course project must fall in the category of "multi-channel online community platforms", and include applications such as "community-based shopping". In that context, examples of useful XML-based services to support these platforms may include synchronized multimedia presentation viewing, and "offline" chat capabilities. A sample specification of an online community platform for a virtual university eBusiness application will be provided later on during the course for illustration purpose.

Generic Architecture Blueprint + Architecture Design Methodology + Mgmt

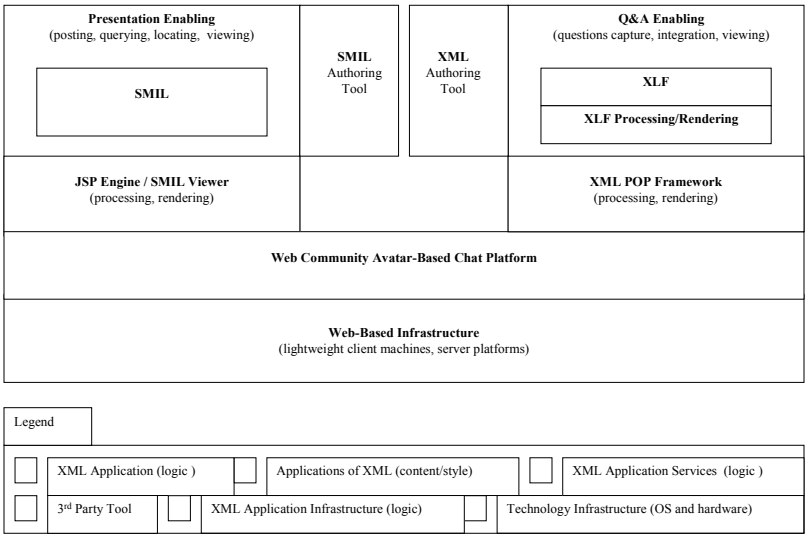






Sample Conceptual Architecture Diagram

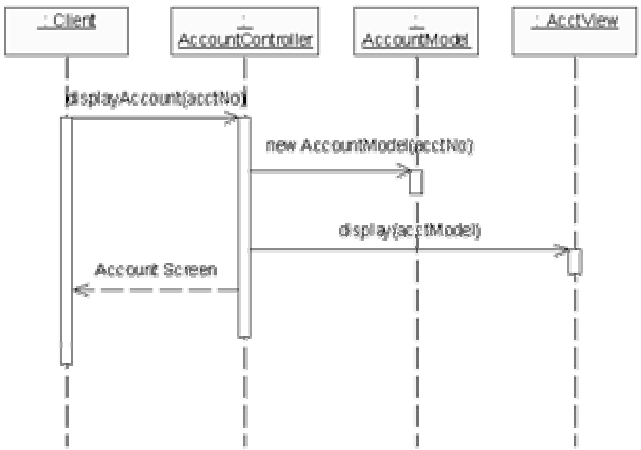
(e.g., virtual classroom environment)



5. MVC Review

http://java.sun.com/blueprints/patterns/index.html

- MVC architecture decouples the code to handle user actions (controller), the data and business logic (Model), and the presentation (View)

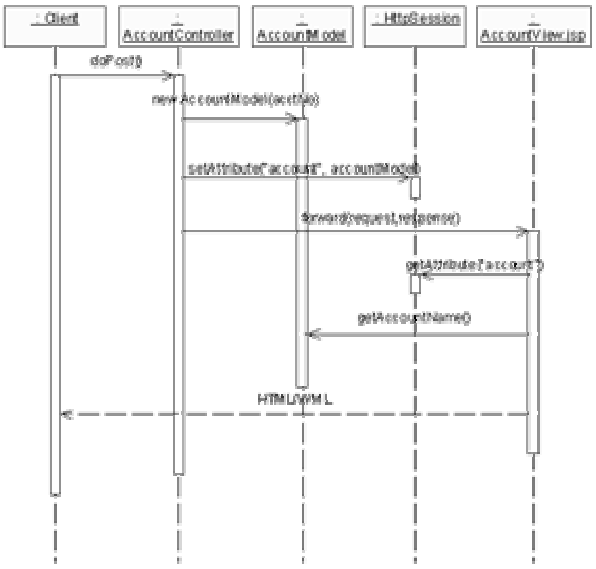


Implementing the "V" of "MVC" Using JSPs

- When the view is implemented as a JSP, the controller object (e.g., servlet) forwards processing of the request and the response to a JSP view
- Controller adds a reference to the model object to the user's session or request object
- JSP gets a handle on the model object and constructs the HTML or other markup to be returned to the client

Implementing the "V" of "MVC" Using JSPs

(continued)

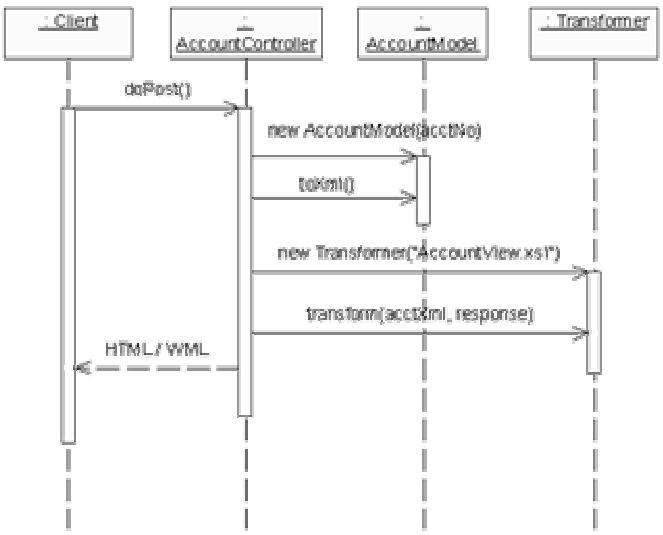


Implementing the "V" of "MVC" Using XSL

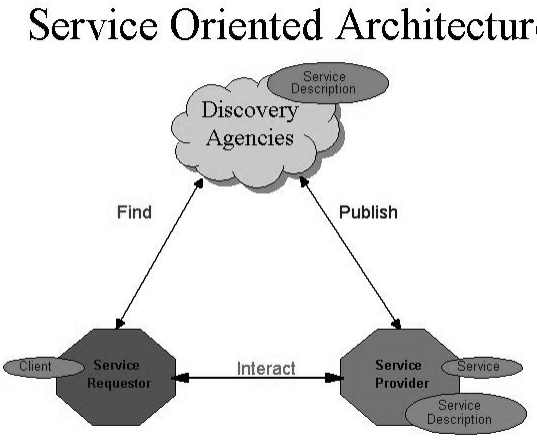
- When the view is implemented in XSL, the basic flow of the transaction remains the same
- The model is represented in an XML format
- Once the model is built, the controller asks for a stylesheet to transform the XML into the desired rendition markup language
- XSL view may be implemented on the client rather than the server, so the controller may return XML to the client

Implementing the "V" of "MVC" Using XSL

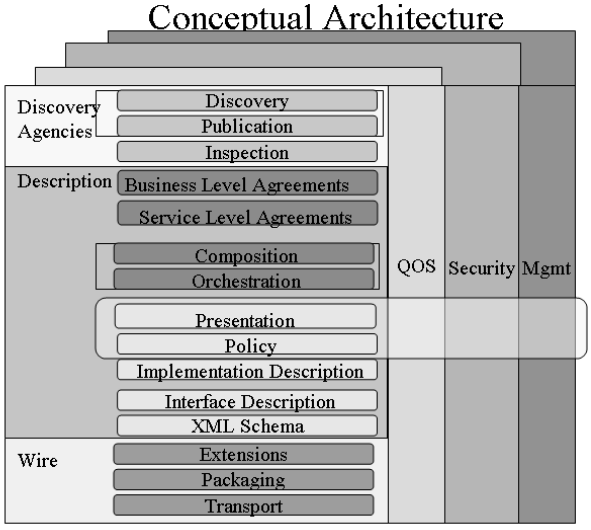
(continued)



6. Service Oriented Architecture



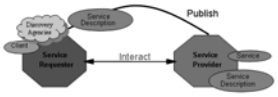
Web Services Stack



Derivative Architecture Patterns

Service Oriented Architecture
Derivative Patterns
Peer to Peer

Service Oriented Architecture
Derivative pattern
Direct interaction



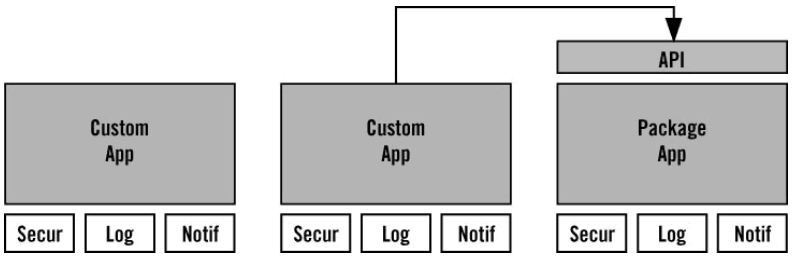
Service Oriented Architecture
Derivative Patterns
intermediary

Service Oriented Architecture
Derivative Patterns
one way message

Towards Web Services

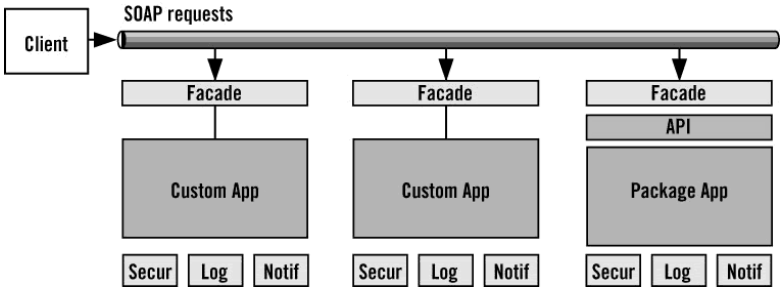
- Plateaus of adoption
  - Integration as an afterthought
  - Web services façades
  - Managed Web services and, finally
  - Paradigm shift
- Industry currently focuses on the second plateau

Integration as an Afterthought



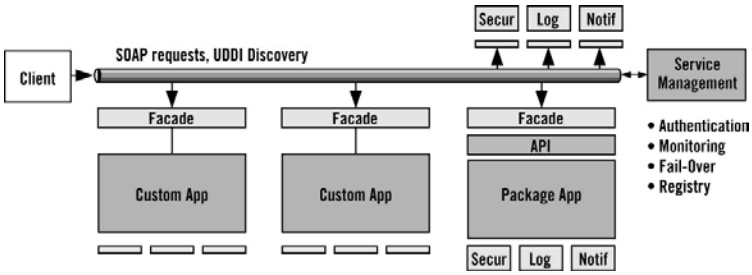
- The current enterprise conjecture consists of a collection of self-contained custom or packaged applications
- Packaged applications may expose functions via an API allowing some level of point-to-point integration

Web Services Façades



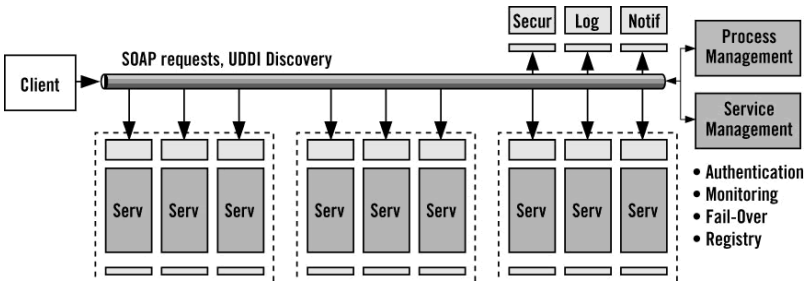
- Adopting Web services first requires "wrapping" existing applications with a Web services façade
- The resulting architecture resembles early EAI implementations, but provides the added benefit of standard protocols

Managed Web Services



- In most cases, package applications are not designed to enable the replacement of underlying services
- As a result, the resulting Web architecture remains a hybrid in which some applications leverage common infrastructure services while others access their own internal services

Paradigm Shift



- In a true service-oriented architecture, all business services use a common set of business-neutral services for logging, notification and security

Challenges

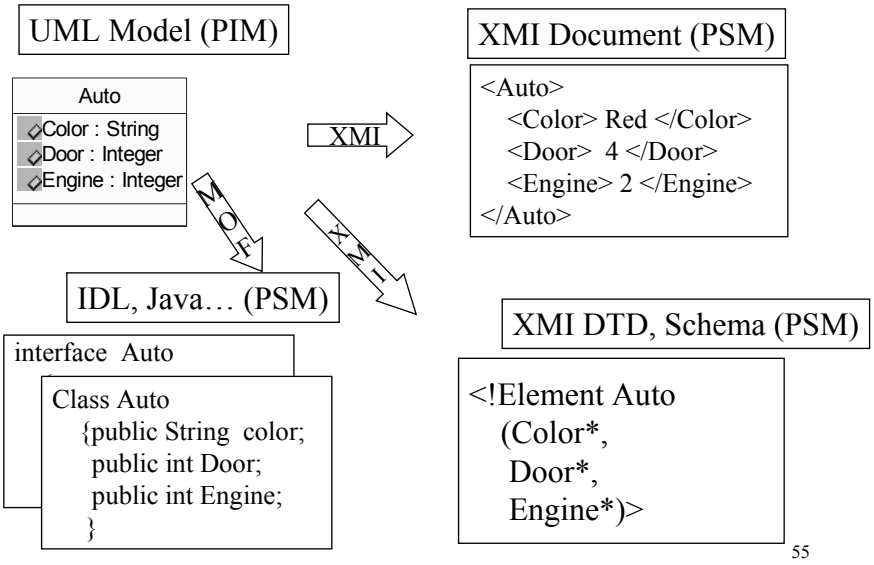
- Evolving standards
- Immature tools
- Semantic mapping
- Network reliability
- Performance
- Application ownership
- Learning curves

7. MDA

- OMG's MDA (Model-Driven Architecture) specification describes:
  - a PIM - Platform-Independent Models (i.e. business design)
  - PSMs - the mapping of a PIM to one or more Platform-Specific Model
- MDA => Model Once, Generate Everywhere
- Review MDA presentations:
  - **http://www.io-software.com**

MDA

(continued)



8. Aspect-Oriented Programming

http://www2.parc.com/csl/projects/aop

- Technology for separation of concerns (SOC) in software development
- AOSD makes it possible to modularize crosscutting aspects of a system such as
  - Design or architectural constraints
  - Systemic properties or behaviors (e.g., logging and error recovery)
  - Features
  - etc.

Example: AspectJ

http://aspectj.org

- A seamless aspect-oriented extension to Java
- Enables the modular implementation of a wide range of crosscutting concerns
- Compilers
  - AspectJ (www.aspectj.org)
  - HyperJ (www.alphaworks.ibm.com/tech/hyperj)

AspectJ Example

http://www.voelter.de/data/articles/aop/aop.html

```
01 aspect DataLog {
02   advise * Worker.performActionA(..), *   Worker.performActionB(..) {
03     static after {
04       if ( thisResult == true )
05         System.out.println( "Executed "+thisMethodName+
06         "successfully!" );
07       else
08         System.out.println( "Error Executing "+
09         thisMethodName );
10     }
11   }
12 }
```

AspectJ – Locking Sample

- Creating a new aspect called lock with introduce and advise cross-cuts

```
01 aspect Lock {
03    Data sharedDataInstance;
04    Lock( Data d ) {
05    sharedDataInstance = d;
06    }
08    introduce Lock Data.lock;
09
10    advise Data() {
11              static after {
12              thisObject.lock = new Lock( thisObject ); }
14    }
17 }
```


AspectJ – Locking Sample

(continued)

- Advising classes that work with the data (note that all the locking code is included in an aspect!)

```
15     boolean locked = false;
16
17    advise Worker.perform*(..), AnotherWorker.perform*(..) {
18          before {
19            if ( thisObject.sharedDataInstace.lock.locked ) // enqueue, wait
20            thisObject.sharedDataInstace.lock.locked = true;
21          }
22          after {
23            thisObject.sharedDataInstace.lock.locked = false;
24          }
25    }
26 }
```

9. Refactoring

http://www.refactoring.com/

- Technique to restructure code in a disciplined way
  - Small code changes (a.k.a., refactorings) are applied to support new requirements and/or keep design as simple as possible
- Enables programmers to safely and easily evolve their code to fulfill new requirements or improve its quality
- Refactoring is a fundamental coding practice of XP and is orthogonal to Agile Modeling, which does not address programming-related issues
- See Java refactoring guidelines at
  - http://www.cafeaulait.org/slides/javapolis/refactoring/
- Refactoring tools:
  - Eclipse supports renaming refactorings that allow you to rename a compilation unit, type, method, field, or parameter
  - Other refactorings allow you to move code, extract methods, and self encapsulate fields

Design Patterns and Refactoring

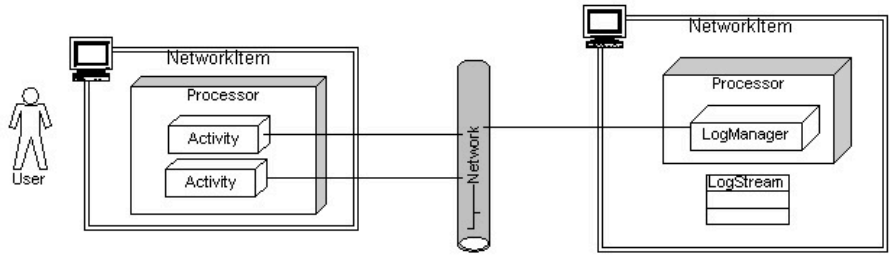
- Refactoring improves code design without adding new behavior
- A design pattern is the description of a design problem and of its solution, which comes with certain benefits and liabilities
  - See http://cs.wwc.edu/~aabyan/PATTERNS/
- Do design patterns drive refactoring or are design patterns discovered in the refactoring result?
  - See Refactoring to Patterns http://www.industriallogic.com/papers/rtp016.pdf

10. Structured Applications Design Tips

- Reuse: should focus on Domain Models/System Family Architectures
- Applications should separate the various information elements (i.e., content, logic, style, and architecture/handling schemes)
- Various content formats: presentation, message, storage, etc.
- Application architecture supports:
  - Web Enabling (WE), XML Enabling (XE), Data Enabling (DE), Enterprise System Assurance Enabling (ESAE)
- Various application support services to support:
  - Interactions with users via content (content + logic) - WE
  - Encoding of user requests as secure (portable) messages (content generation) - XE/ESAE
  - Processing of user requests via logic (content + logic) - XE
  - Rendering of content via logic using style (content + style + logic) - WE/XE
  - Querying information via logic (content + logic) - XE/DE
  - Interactions with back office via content (content + logic) - XE/ESAE

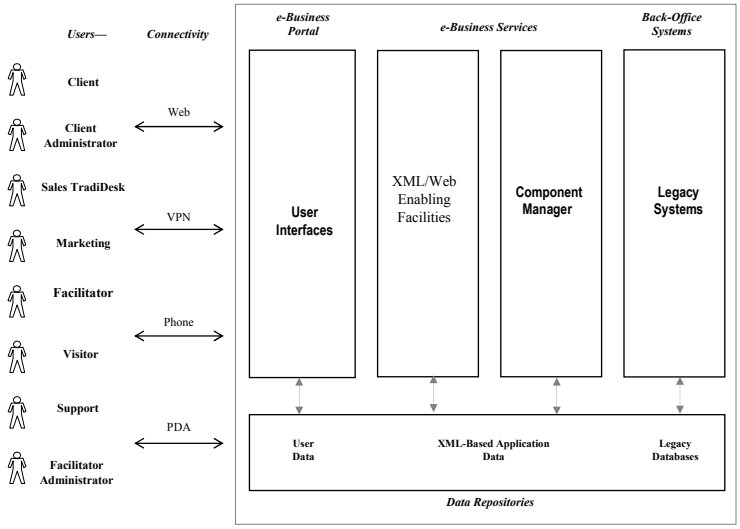
Investigating Logging Infrastructure

(e.g., virtual classroom environment)

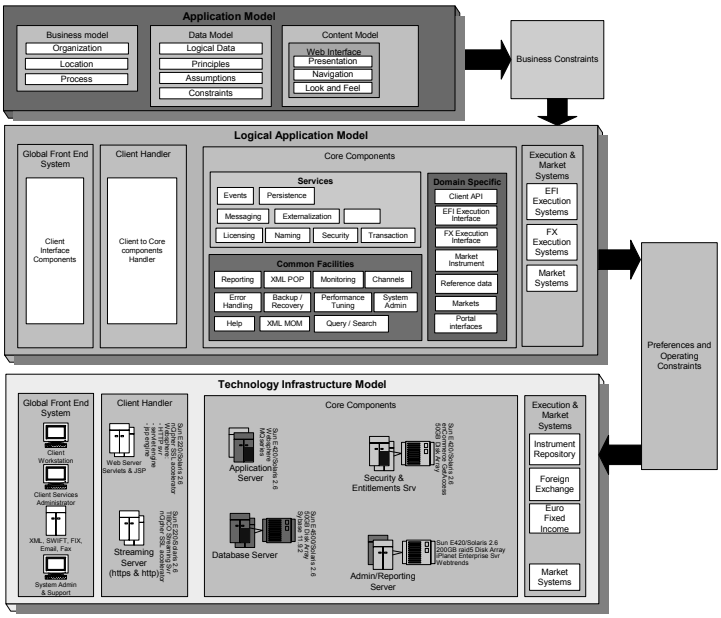


Refined Application Architecture Blueprint

(e.g., virtual classroom environment)

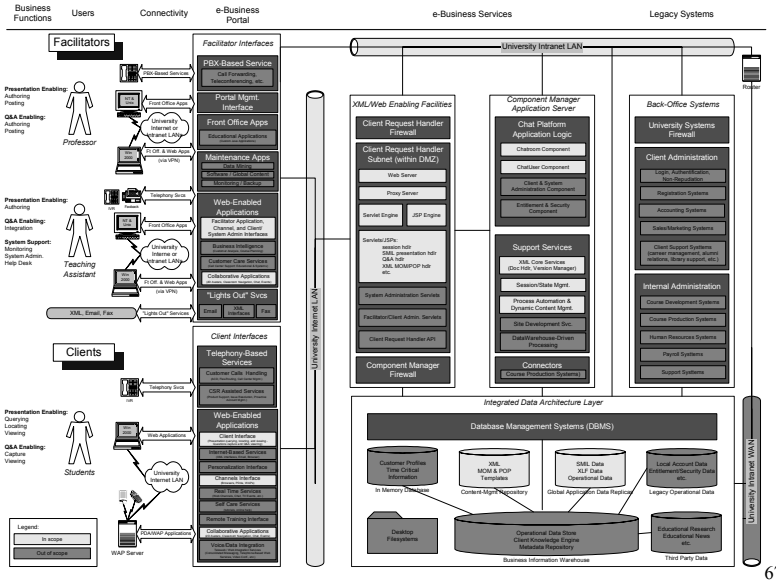


Mapping Application to App. Infrastructure



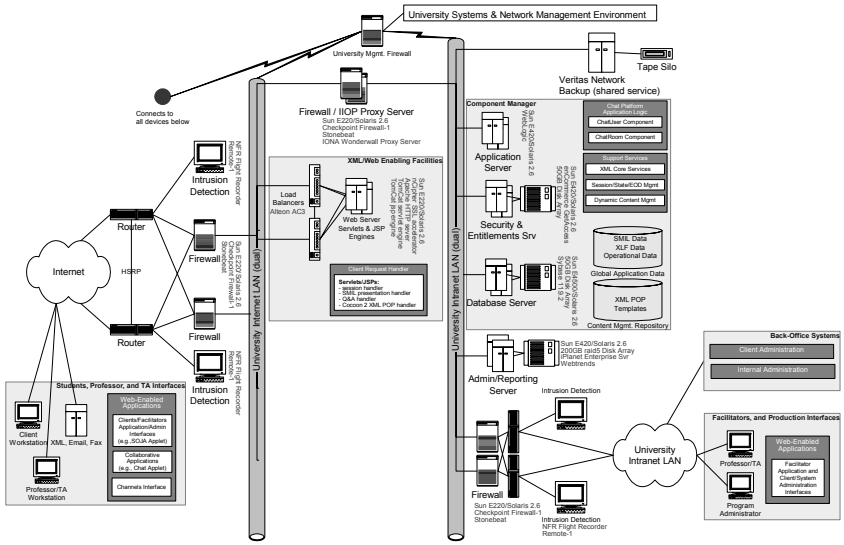
Sample Logical Architecture Diagram

(e.g., virtual classroom environment)



Sample Logical Architecture Diagram

(e.g., virtual classroom environment)



Part IV

Architectural Principles

What is “Architecture”?

- The art or practice of designing structures
- Formation or construction as part of a conscious act
- Architectural product or work
- A method or style of building
  - [Webster's Dictionary]

Definition

"The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them" - [Bass et al 2003]

Software Elements

- Architecture defines *Software Elements*
- Defines how elements interact
- Elements interact by means of public interfaces
- The private aspects of an element are the province of design and implementation
- Architecture suppresses information that does *not* relate to how elements use, are used by, or relate to other elements

Multiple Structures

- There is no definitive structure that defines an architecture
- Structures can be used to:
  - Define static partitioning and assign functionality
  - Represent snapshots of dynamic behavior
  - Support allocation such as process to processor mappings

It's Everywhere! It's Everywhere!

- *Every* system has an architecture
- It can always be reasoned about as external interaction between elements
- However the architecture may be obscured, or simply not known
- It is possible for architecture to exist independently of its documentation!

Its Not Just Boxes & Lines

- Static diagrams are typically passed off as architecture
- The definition of architecture includes both static and dynamic structures
- Architecture includes defining element behavior insofar that it is part of the visible inter-element process of interaction

The Good, The Bad & The Ugly

- The definition of architecture is indifferent to whether it is good, bad, or simply adequate
- An architecture that prevents the system from meeting its functional and non-functional requirements is still an architecture!

Why is Architecture Important? (1)

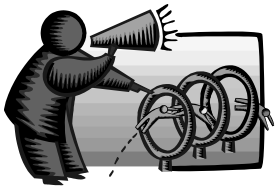
- Intellectual Control and Complexity Management
  - ✓ It lets you establish a strategic framework to provide a holistic view of the system
  - ✓ It provides a foundation for recording all the "tactical decisions" made in each iteration
  - ✓ It provides a "reference" for future enhancements
  - ✓ It facilitates a basis for communication by establishing a common vocabulary for strategic and tactical design decisions

Why is Architecture Important? (2)

- It is an effective basis for large-scale reusability
  - ✓ Partitioning identifies opportunities for "design with reuse" activities, these represent
    - ✓ What abstractions already exist that can be leveraged in the emerging architecture?
  - ✓ Architecture facilitates "design for reuse" activities
    - ✓ Are there components we can extract and reuse in future projects?
    - ✓ Have we designed a lar" things before?

Why is Architecture Important? (3)

- Project Management
  - ✓Resources and personnel can be organized along component lines
  - ✓A team can be allocated to produce one or more subsystems
    - ✓The architecture defines the basis of integration at the interface level



Architecture and Software

- An abstract representation of a systems components and behaviors
- Represents a solution synthesis, addressing both functional and non-functional requirements
- It typically does not include implementation details, that behavior is addressed by the developer
- Architecture may be:
  - Product
  - Style

Architectural Types

- System Architecture
  - Architecture as a *Product*
- Reference Architecture
  - Architecture as a *Method* or *Style*

System Architecture

- Architecture as *product*
- Addresses the identification of:
  - Components
  - Component Interfaces
  - Component Interactions
  - Component Constraints
- Provides a high level static and dynamic model to be used as a basis for detailed design and implementation
- Typically specified as a UML model (or something more informal for customer consumption)

Architectural Standards

- Establishes vital system components and constructs that need to be retained when new features are added
- Violating these standards prevents the architecture adapting gracefully in the presence of change

System Views

- System architecture can be represented using multiple views
- Each view emphasizes a specific aspect of the system
- Descriptive Views
  - A formal arrangement of design elements
  - Typically used to illustrate to the customer that requirements are being met
- Prescriptive Views
  - Specifies how the system will be built

Technology Influences

- Architecture and Technology have a basic synergy
- Technology is often an enabler for certain types of system architecture
- N-Tier Internet applications - rely on browser standards, application servers, distributed protocols, fast networking capabilities … not possible without these technology components in place

Reference Architecture

- An architectural style or method
- The architect selects elements from a reference architecture and uses them as a basis for producing a given system architecture
- For example
  - J2EE is a reference architecture that provides support for component abstractions, such as Servlets or EJB, and a range of abstract services for managing concerns ranging from persistence to transaction management

Reference Elements

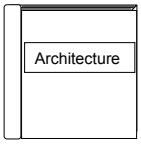
- Defines a standard terminology
- Provides standard template components
- Describes the responsibilities of basic abstractions (e.g. session versus entity EJB's)
- May support a development methodology

Advantages

- Standard, unified terminology and concepts
- Provides simple, easy to use abstractions
- A proven reference architecture transfers quality and reliability to subsequent system architectures
- Higher degree of reusability across projects
- Traceability from architecture through to implementation

Reference Architecture Specification

- Organization-wide standards and guidelines
  - Interface specifications
  - Use of frameworks
  - Use of design patterns
- Must be rigorously enforced
- $$ Investment required



Architecture and Design

- Architecture
  - Operates at a higher level of abstraction than design
  - Emphasizes structuring principles, partitioning subsystems, assigning functionality to components, specifying interfaces, communication and concurrency protocols, process to processor allocation, non-functional capabilities….
- Design
  - Addresses detailed concerns required for implementation
  - Interface signatures, I/O, algorithms, data structures
  - Represents a detailed refinement of an established architecture

Key Characteristics

- Abstraction
- Encapsulation
- Partitioning and Decomposition
- Layering
- Views
- Capabilities

Abstraction

- Simplifies the complicated
- Separates relevant characteristics from irrelevant
  - Selectively ignores some details in favour of others.
- Allows us to 'see the wood for the trees'
- Data Abstraction - create an interface to some data which is not dependent on the storage of that data.
  - In a list "*getNextItem*" is more abstract than "*add 1 to index; read(index*)"
  - "Strong black coffee, no milk" is more abstract than "number 52, please"
- 'Vehicle' is more abstract than 'Car'.

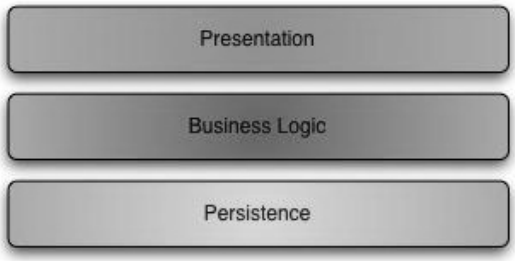
Encapsulation

- Distinguish what an object does from how it does it
  - interface vs. implementation; public vs. private; external vs. internal
- An object supports:
  - 'visible' operations which form its external interface
  - 'hidden' data and operations defined inside the object boundary
- Operations provide high-level services
- Objects invoke each others' public operations
  - Each class is simpler
  - Can change internals with no impact
  - Avoids code duplication
  - Better integrity
  - More maintainable

Partitioning and Decomposition

- The main structuring principle is "divide and conquer'
- Strategies
  - *Decomposition* : Break the system down hierarchically into compositional structures, moving from the general to the specific
  - *Partitioning* : divide the software into units, such as functions, modules or classes at a given level of abstraction

Layering



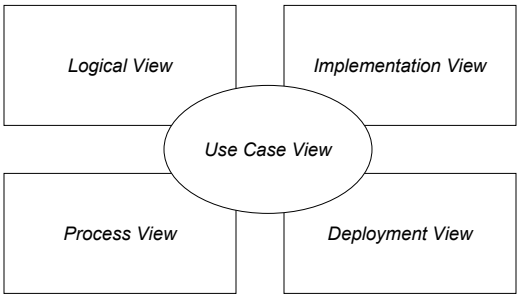
Layering (2)

- Example Strategies:
  - Layering By Generality - domain independent layers at the bottom of the "layer stack", domain-specific at the top
  - Layering by Abstraction - specific "small grain" detailed functionality at the bottom of the stack, high level services at the top

Architectural Views

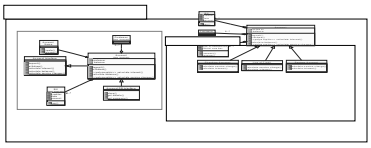
- Each role requires a "view" of the system designed to emphasize certain relevant characteristics and de-emphasize or remove others
- A **View** is an abstraction of the system from a particular perspective, it contains:
  - ✓ A description of the vantage point – what concerns does it emphasize and to which group of stakeholders is it relevant?
  - ✓ View Elements – what elements and their relationships does it document?
  - ✓ Organization – how is the view structured?
  - ✓ How is it interconnected or interrelated to other views?
  - ✓ How is the view created?

The "4 + 1" View Model



The Logical View

- An abstraction of the design model
- Emphasizes functional requirements
  - ✓The view is constructed using *major* design packages, subsystems and classes



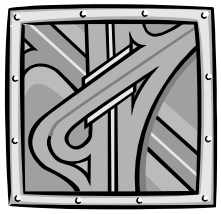
A Logical View

The Implementation View

- Describes the organization of static software "modules"
  - e.g. source code, data files, components
- Describes packaging and layering
- Addresses configuration management
  - E.g. the release number, the business and development owners

Process View

- Addresses dynamic and concurrent behavior
  - ✓Activities of processes and/or lightweight threads
  - ✓Deadlock, livelock, race conditions, mutual exclusion
  - ✓Scalability
  - ✓Performance
  - ✓Fault Tolerance

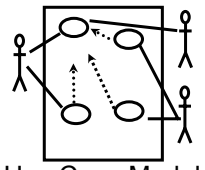


Deployment View

- Describes how runtime components are mapped to the underlying hardware and software environment
- Addresses issues such as deployment, installation, system-level optimization and performance

Use-Case View

- A “binding” view, used to drive and validate all other views in the model
- Initially used to formulate other views
- Later, scenarios will be used to validate and test these views



Architectural Capabilities

- Non-functional, observable system characteristics
  - Availability
  - Reliability
  - Manageability
  - Performance
  - Scalability
  - Extensibility
  - Validity
  - Reusability
  - Security

Availability

- The degree to which a system is operable when called upon at an unknown or random time
- Expressed as a ratio of:
  - 1 minus *unavailability* e.g. 0.965
- Applications often require a degree of "downtime" for maintenance, bookkeeping etc, typically governed by an SLA (Service Level Agreement)

Reliability

- Ability of a system to perform its required functionality under stated conditions for a given period of time
- Typically measured as MTBF (Mean Time Between Failure)

Manageability

- The set of services required to ensure the continued correctness and integrity of the architecture
  - e.g. security and server management
- May be addressed by the reference architecture (e.g. JMX for Java/J2EE systems)

Performance

- Ability of a system to execute functions within a given temporal "window"
  - Not necessarily running "fast enough"
- Architect should target performance criteria before implementation
  - Architect required to make "informed estimation" regarding performance of models
- Two key measurements
  - Response Time
  - Response Ratio (response time to time it actually takes to execute the function)

Scalability

- Two basic types of hardware scalability
  - Horizontal, adding new servers to distribute load
  - Vertical, adding more CPUs and memory to handle a greater load

System Scalability

- A system can scale to handle more users and transactions in the following ways:
  - Throw more hardware at the problem!
  - Improve communication efficiency e.g. pool communication resources, make a single instead of multiple calls to the same distributed component…
  - Transparent load balancing across application servers / application instances
  - Efficient resource management, connection pooling, caching

Scalability Characteristics

- Capability to dynamically add more hardware capacity without incurring downtime
- The provision for “gracefully degrading” functionality up to 100% capacity
- The ability to suspend less critical work at times of peak load

Extensibility

- Capability of an architecture to be extended to facilitate the graceful addition of new requirements

Validity

- Also known as testability
- Architect must establish empirical and repeatable criteria for what constitutes a valid system
- Validity tested at boundaries between system layers
  - Presentation to Business Logic Layers
  - Business Logic to Persistence Layers

Reusability

- Two key factors to address:
  - Design *for* Reuse, genericizing and packaging software for reuse in multiple contexts
  - Design *with* Reuse, leveraging existing reusable software as part of an emerging system architecture
- Things to reuse : Code, Frameworks, Interfaces, Patterns, Heuristics, Design Artifacts, Analysis Models, Requirements…

Security

- Protection of resources and assets from loss:
  - *Privacy*, preventing disclosure to unauthorized persons
  - *Integrity*, preventing illegal corruption or modification of data
  - *Authenticity*, has the system correctly identified a user? Has data been transmitted successfully?
  - *Availability*, are my services all available for utilization?

Security Services

- Standard security services include:
  - Identification and authorization
  - Access control
  - Accountability and auditing
  - Data confidentiality
  - Data integrity and recovery
  - Secure data exchange
  - Non-repudiation of data origins and delivery
  - Reliability
- The implementation of these services are defined by a *security policy*

Part V

Use Case Driven Development

Learning Objectives

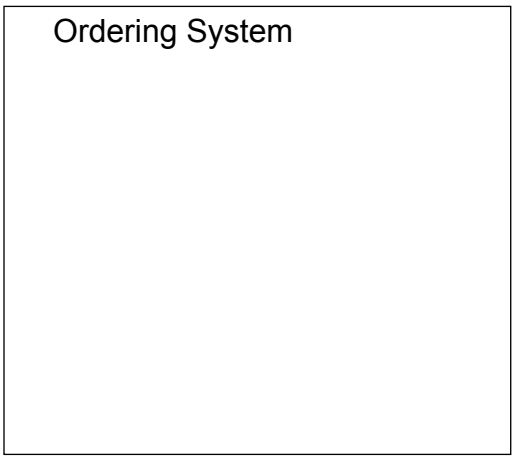
- Introduce use case modelling and specification
- Outline the basis of use-case driven development
- Distinguish between ***Real*** and ***Essential*** modeling
- Examine the use case driven process from requirements through to test

What is a Use Case?

- A Use Case is a modelling entity used to describe the functional requirements of a system.
- A Use Case describes a "*complete functionality*". This is one general usage of the system.

"*A description of a set of sequences of actions, including variants, that a system performs that yields an observable result of value to an actor*" [Booch 1999]

Boundary Notation

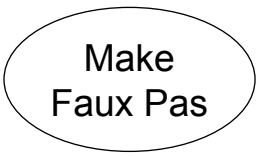


System Boundaries

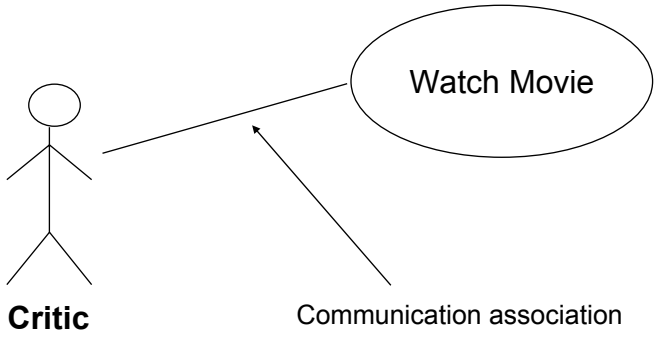
- This simple notation describes the *boundary* of the system.
- Note that the system may not be a software system; it could be a business or a hardware device for example.
- Determining the boundary can be a design activity.
- Note also that for simple systems, showing a *system boundary* might not be necessary.

Notation

- A Use Case is simply represented as an "oval", annotated with a name.



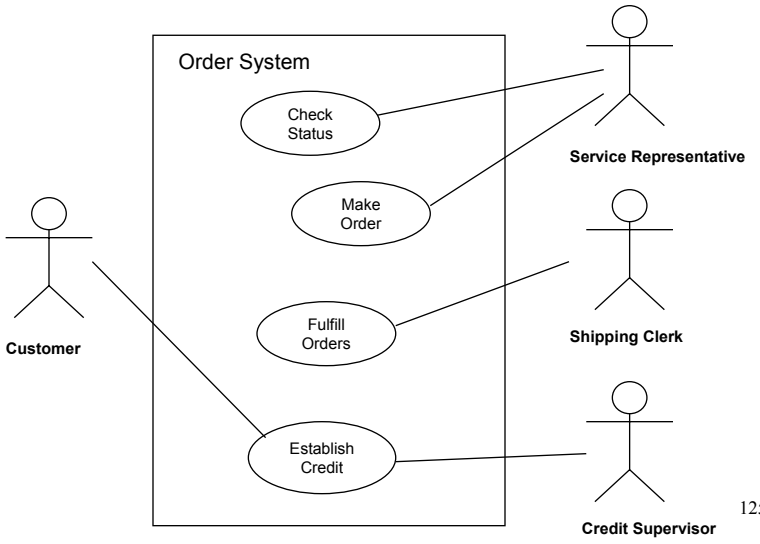
Communication Notation



Communication Semantics

- The communication may be directional (using an arrow) or bi-directional
- The relationship may hide extremely complex patterns of interaction between actor and use case!

An Example Use Case Model



Identifying Use Cases

- What functions will a given actor want from the system?
- What measurable value does an actor require from the system?
- Given an actor role, what sort of business event may they wish to initiate?
- Does the system use transient or persistent data? What actors will Create, Read, Update and Delete that data?
- What notifications does the system send to its actors?

Use Case Templates

- The real detail is not shown on a use case diagram
- A use case diagram must be supplemented with a use case description.
- The use case descriptions are usually created in a standard format.
- There are various published templates to follow for this activity.
- If you adopt this technology you will have to adopt a template or develop your own.

What do we need to capture?

- Goals
- Preconditions and Postconditions
- Initial Description
- Scenarios
- Non-functional requirements

Goals

- A use case should have a well defined *goal*, and one or more *backup goals*.
- An actor has a set of *responsibilities*.
- To fulfil those responsibilities it formulates *goals*.
- These goals are carried out by *actions*.

Why Goals?

- Goals tend to be tracked informally – responsibilities are typically mapped directly to actions
- At the use case level we can set a goal that will
  - Meet one or more actor responsibilities
  - Are realized by the scenarios within the use case
- Goals are reflected in the use case name, and a specific goal-capture section.

Goal Example

- A Customer actor has a responsibility to make and pay for an order.
- A use case may have the goal "allow Customer to make electronic payment"
- The use case will include a flow of events (scenario) to meet that goal.

Preconditions and Postconditions

- No use case is an island. It may have dependencies on other use cases that change system state.
- **Precondition** – what system state must be true in order for the use case to execute?
- **Success Postcondition** – what will be the state of the system after successful use case completion?
- **Failure Postcondition** - what will be the state of the system after unsuccessful use case completion?
- These are *predicates*, based on system state that all stakeholders should be able to understand

Scenarios

- These represent the actual “meat” of a use case specification
- A scenario is a flow of events demonstrated by a given *use case instance*.
- Two types:
  - Primary Scenario
  - Secondary or Variant Scenarios

Primary Scenario

- The primary scenario represents a flow of events where everything goes as expected.
- It is also known as the “Happy Day Scenario”.
- A use case typically has a single Primary Scenario.

Secondary Scenarios

- These represent a flow that deviates from the Primary Scenario.
- This includes:
  - Less probable decisions based on state or use input
  - Incomplete or erroneous user input
  - Exceptional conditions in the system or environment
- These are typically invoked as **variation points** from the primary scenario.

Use Case Detail Levels

- While use cases can be used in different ways there are two basic levels of detail
- *Essential* – to capture the essence of the requirements in a usage centered, technology and implementation-independent manner.
  - Also known as *Abstract* or *Business* use cases.
  - Designed to meet non-technical stakeholder needs
- *Real* – to capture the behavioural requirements in a fashion that references, user interface components, implementation details and technical constraints
  - Also known as *Concrete* or *System* uses cases
  - Designed to meet technical stakeholder needs.

Essential Scenario Example

Ordering a Pizza Online Workflow

- The customer inputs a pizza crust, classic, deep pan or stuffed.
- The customer inputs a base size of 10", 15" or 17"
- The customer selects one or more toppings from our daily selection.
- The system calculates the combined price of crust, base and topping(s).
- The customer submits the order to the system
- The system verifies if the combination of crust, base size and toppings is legal (according to Business Rule #5)
- If the order is legal the system will return a confirmation and delivery time.
- If the order is not legal the system will ask the user to rebuild their pizza go to (1).

Real Scenario Example

Ordering a Pizza Online Workflow

- The customer selects a pizza crust, classic, deep pan or stuffed from a drop down menu.
- The customer selects a single checkbox indicating a base size of 10", 15" or 17"
- The customer customer goes through an interative process, adding toppings dynamically from our daily selection. This will *not* result in a page submit and should be done using a scripting language directly on the page.
- The system dynamically calculates the combined price of crust, base and topping(s), again using a page-centered scripting language.
- The customer presses the "Process Order" submit button.
- The system verifies if the combination of crust, base size and toppings is legal (according to Business Rule #5)
- If the order is legal the system will return a confirmation and delivery time.
- If the order is not legal the system will ask the user to rebuild their pizza go to (1).

Compromise

- The favored approach is to maintain *two* different sets of documentation.
- This has negative implications for time, budget and maintainability.
- A *possible* compromise is to merge three levels of detail into a single document.

Compromise Example

- ***Example Use Case #1***
- *Description* – this should provide a comprehensive executive summary of the purpose, focusing on *what* it will do and what results of value it will provide to the Actors.
- ***Workflow***
  - **Provide an essential description of the step**.
  - *Follow it with any required real information*.

How much Depth of Detail?

- What *depth* of detail do I need to specify my use cases at?
- ***External or "White Box" Detail***
  - Focuses on activities that are directly visible to the actor
  - The system is a black box that simply produces outputs
- ***Internal or "Black Box" Detail***
  - Focuses on both user and system level requirements.
  - Opens up the system focusing on *What* versus *How*.

White Box Specification

- ***Advantages***
  - Does not make a premature commitment to design detail.
  - Does not require the analyst to understand the underlying object model.
  - Allows the analyst to focus on user interaction not implementation detail.
- ***Disadvantages***
  - It will not completely model the system requirements. Who will fill in the gaps?
  - Difficult to validate user-centric requirements against an emerging object architecture

Black Box Specification

- ***Advantages***
  - Teases out those hidden system requirements that are not visible to the user
  - Gives guidance on how to partition the emerging object architecture
- ***Disadvantages***
  - Use Cases are not objects – this may lead to functional decomposition!
  - Users may not understand how objects and implementation mix

Use Case Based Requirements

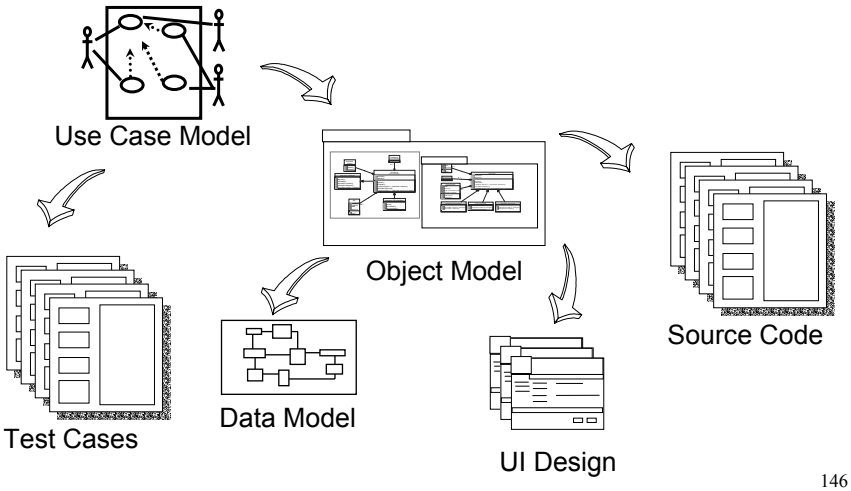
- The use case model is an artifact of the requirements process
- It elicits *what* the system should do from the users point of view
- The model represents a common basis of communication between developers, management, stakeholders and user
- It determines:
  - What to build.
  - When to build it (through use case prioritization)

Requirements to Code : Problems

"*two cultures divided by a common language*"

- Requirements often address "real-world" abstractions and items – this does not integrate well with an algorthmic view of the world.
- A given requirement may be distributed through an architecture, rather than centrally located.
- Requirements may often address issues such as performance, which come from good design practice but are not enforced algorithmically.
- System design isnt purely driven by user requirements! What about constraints? What about system requirements?

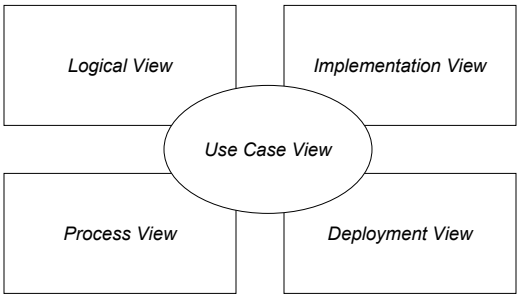
Use Cases and Development



Architectural Views

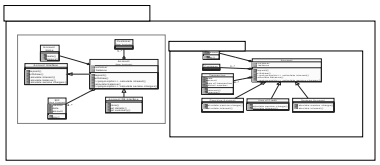
- Each role requires a "view" of the system designed to emphasize certain relevant characteristics and de-emphasize or remove others
- A **View** is an abstraction of the system from a particular perspective, it contains:
  - ✓ A description of the vantage point – what concerns does it emphasize and to which group of stakeholders is it relevant?
  - ✓ View Elements – what elements and their relationships does it document?
  - ✓ Organization – how is the view structured?
  - ✓ How is it interconnected or interrelated to other views?
  - ✓ How is the view created?

The "4 + 1" View Model



The Logical View

- An abstraction of the design model
- Emphasizes functional requirements
  - ✓The view is constructed using *major* design packages, subsystems and classes



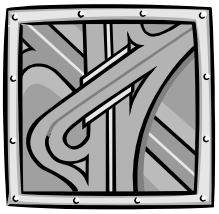
A Logical View

The Implementation View

- Describes the organization of static software "modules"
  - e.g. source code, data files, components
- Describes packaging and layering
- Addresses configuration management
  - E.g. the release number, the business and development owners

Process View

- Addresses dynamic and concurrent behavior
  - ✓Activities of processes and/or lightweight threads
  - ✓Deadlock, livelock, race conditions, mutual exclusion
  - ✓Scalability
  - ✓Performance
  - ✓Fault Tolerance

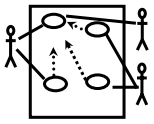


Deployment View

- Describes how runtime components are mapped to the underlying hardware and software environment
- Addresses issues such as deployment, installation, system-level optimisation and performance

Use-Case View

- A “binding” view, used to drive and validate all other views in the model
- Initially used to formulate other views
- Later, scenarios will be used to validate and test these views



Use Case Model

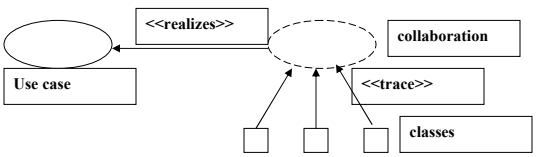
Use Case Driven Development

- Use cases can, and are used to drive forward the software development process
- They are a common point of reference and agreement between stakeholders.
- Use cases specify the functional requirements from a user-centered perspective, addressing the functional behaviours they actually require
- They provide a ideal basis for classic validation and verification (V&V) activities, against requirements:
  - Validation : Are we building the right system?
  - Verification : Are we building the system right?

V&V Activities

- ***Validation***
  - A use case model is *realized by* a design model
  - A use case model is *implemented by* an implementation model
- ***Verification***
  - A use case model is *verified by* a test suite.

Analysis and Design



- Use cases are realized using Object-Oriented architectural models
- The use case maps to a conceptual construct called a *Collaboration*.

Collaborations

- A collaboration is a UML™ construct that represent a "society" of classes, objects, packages and other modeling elements that realize the use case.
- The collaboration encapsulates:
  - ***Static modeling elements*** – these specify how the system is actually partitioned.
  - ***Dynamic modeling elements*** – this specifies how the architecture behaves in order to meet the requirements specified in the use case.

Collaboration Issues

- Paradigm Mismatch : use cases are primarily functional requirements, how does this map cleanly to an object model?
- Inexperienced analysts/designers may translate use cases into systems with "functional stripes" not object-oriented ones!
- Requires a body of guidelines to facilitate and guide the mapping process
- e.g. CRC cards, Anthropomorphism, Brainstorming, Verb-Noun finding

Implementation

- The object architecture is then mapped to executable artifacts that will compose the actual running system
- Typically there is a strong correspondence between the object architecture and the code (and hence traceability back to the use cases).
- Other issues that may also effect the mapping:
  - Deployment configuration
  - Vendor and Environmental Constraints
  - Performance, Capacity, Security, Availability

Testing

- Use cases drive the development of functional test cases
- A single use case is used to create a *suite* of **test cases**
- Each scenario maps to a specific test case:
  - The primary scenario represents the "ideal outcome", the probable path of success
  - This is typically modelled as a **positive test case**
  - The secondary scenarios capture variant, exceptional and error based event flows
  - These may be used to model variant positive or **negative test cases**

Test Case Generation

- To generate a test case, we reason specifically about use case *instances* (i.e. a specific behavioural path through the use case type)
- Type-based information, such as preconditions based on state, and actor input *have to be realized with specific values*.
- All the abstract information in the use case must be realized with actual concrete values (using classic data ranging techniques) and behaviour from the implementation model.

Stepwise Testing

- Each step in a test case, corresponds to a step in a scenario
- A step, when used in a test procedure is typically assigned a value of:
  - ***Pass*** : the step completed as expected
  - ***Warning*** : the step completed but there are results that require further analysis
  - ***Fail*** : the step failed, and the test case was halted at that point

A Simple Test Template

- ***Test Case Name***: <This is a name assigned to a primary or secondary scenario>
- ***Use Case*** : <The originating use case>
- ***Setup*** : <describe the environmental setup steps>
- ***Teardown*** : <describe how to "clean up" the environment after the test>
- ***Step : For Each Step, describe the following***
  - ***Step ID*** : <the test id for the step>
  - ***Description*** : <the behaviour that the step should exhibit>
  - ***Status*** : <Pass | Warning | Fail>
- ***Overall Status*** : <Pass | Warning | fail>

Test Case Interactions

- Testing all the possible interactions of test cases is next to impossible within any normal human lifespan
- Common patterns of interaction (usually determined by preconditions on a use case or shared data in the object model) should be captured
- These can be tested by combining test cases derived from different use cases into a single test suite.

What is Requirements Traceability?

- The ability to track relationships between requirements and their realizations in other parts of the development process such as design, implementation and test cases.
- These relationships are utilized to manage change effects by establishing a traceable relationship between a requirements and its realizations

IEEE Definitions (1990)

- "*The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another; for example, the degree to which the requirements and design of a given software component match."* (IEEE 610.12-1990 §3)
- "*The degree to which each element in a software development product establishes its reason for existing; for example, the degree to which each element in a bubble chart references the requirement it satisfies.*" (IEEE 610.12-1990 §3)

Use Case Traceability

- Use cases "bind" the core workflows : Requirements, Analysis, Design, Implementation and Test, through a "trace" dependency
- A use case in the requirements is traceable to a collaboration of classes in the analysis and design models.
- These in turn are trace to actual implementation components.

Types of Traceability

- There are two main Traceability Types
- ***Explicit Traceability*** – a dependency explicitly established by the development team
  - e.g. "Customer wants to pay with credit card" is an elicited requirement that can be traced to the Make Payment use case.
- ***Implicit Traceability*** – an implicit consequence of an explicit traceability relationship or a consequence of the modeling / development paradigm
  - e.g. A society of uml modeling elements can be traced to a *Collaboration*. A collaboration can be traced to a use case.

Other Traceable Properties

- Action Items
- References
- Assumptions
- Glossary Terms

What Should be Traced?

Here are some key examples:

- Stakeholder requirements to use cases
- Use cases to design and implementation models
- Use cases to test suites
- Implementation models to test suites

Part VI

Extreme Programming and Agile Modeling

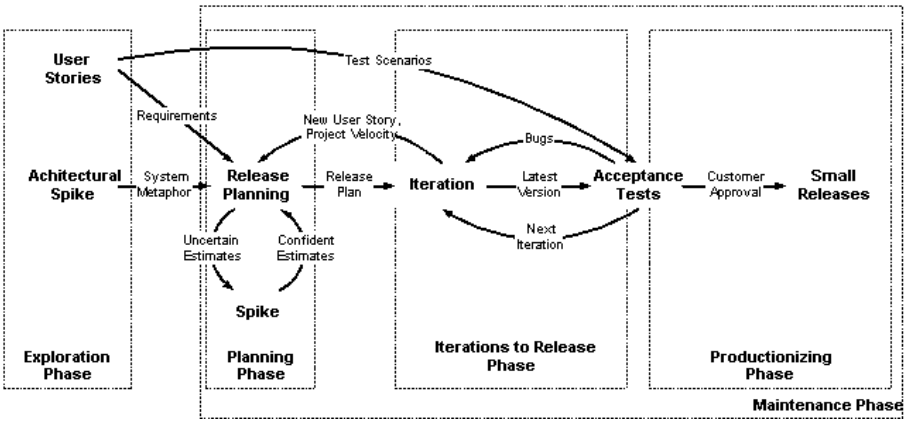
http://www.agilemodeling.com/essays/agileModelingXP.htm

eXtreme Programming (XP)

http://www.extremeprogramming.org/

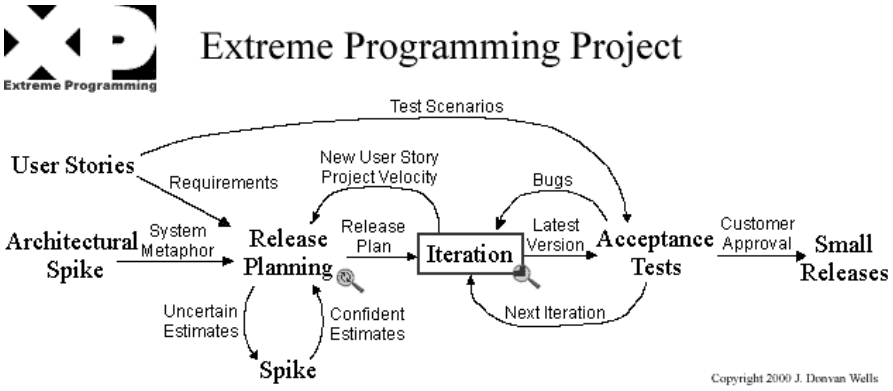
- A lightweight software methodology
  - Few rules and practices or ones which are easy to follow
  - Emphasizes customer involvement and promotes team work
  - See XP's rules and practices at http://www.extremeprogramming.org/rules.html

XP Project Lifecycle

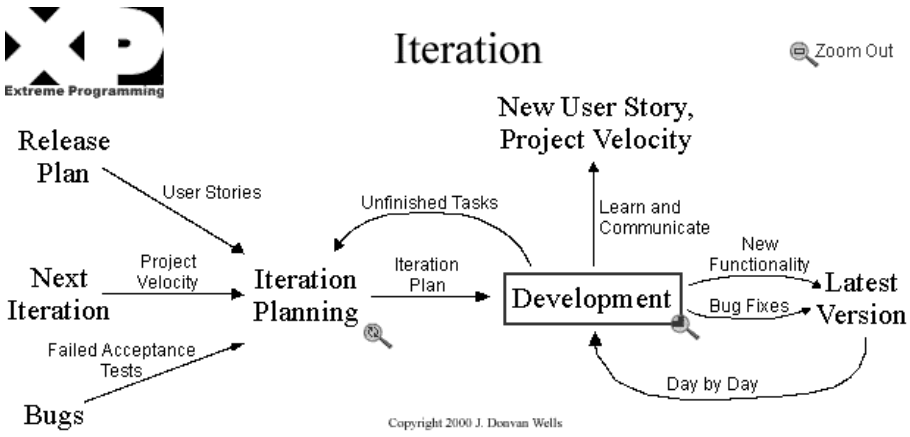


eXtreme Programming (XP) Map

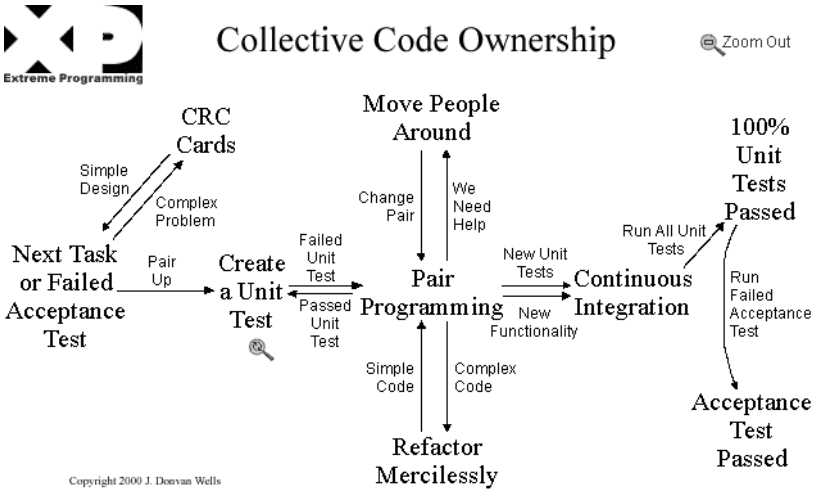
http://www.extremeprogramming.org



XP Iteration Planning



XP Unit Testing



Agile Modeling & XP

http://www.agilemodeling.com/, http://www.agilemodeling.com/resources.htm

- Practices-based software process whose scope is to describe how to model and document in an effective and "agile" manner
- One goal is to address the issue of how to apply modeling techniques on software projects taking an agile approach such as:
  - eXtreme Programming (XP)
  - Dynamic Systems Development Method (DSDM)
  - SCRUM
  - etc.
- Using modeling throughout the XP lifecycle
  - http://www.agilemodeling.com/essays/agileModelingXPLifecycle.htm

Part VII

Agile Software Development

"Agile" Methodologies

- See Agile Project Development Methodology at Work:
  - http://www.thoughtworks.com/library/agileEAIMethods.pdf
  - http://www.thoughtworks.com/library/newMethodology.pdf

Part VIII

Roles and Types of Standards

Standards

- ISO 12207
  - http://www.acm.org/tsc/lifecycle.html
  - http://www.12207.com/
- IEEE Standards for Software Engineering Processes and Specifications
  - http://standards.ieee.org/catalog/olis/se.html
  - http://members.aol.com/kaizensepg/standard.htm

Part IX

Conclusion

Course Assignments

- Individual Assignments
  - Reports based on case studies
- Project-Related Assignments
  - All assignments (other than the individual assessments) will correspond to milestones in the team project.
  - As the course progresses, students will be applying various methodologies to a project of their choice. The project and related software system should relate to a real-world scenario chosen by each team. The project will consists inter-related deliverables which are due on a (bi-) weekly basis.
  - There will be only one submission per team per deliverable and all teams must demonstrate their projects to the course instructor.
  - A sample project description and additional details will be available under handouts on the course Web site.

Course Project

- Project Logistics
  - Teams will pick their own projects, within certain constraints: for instance, all projects should involve multiple distributed subsystems (e.g., web-based electronic services projects including client, application server, and database tiers). Students will need to come up to speed on whatever programming languages and/or software technologies they choose for their projects - which will not necessarily be covered in class.
  - Students will be required to form themselves into "pairs" of exactly two (2) members each; if there is an odd number of students in the class, then one (1) team of three (3) members will be permitted. There may not be any "pairs" of only one member! The instructor and TA(s) will then assist the pairs in forming "teams", ideally each consisting of two (2) "pairs", possibly three (3) pairs if necessary due to enrollment, but students are encouraged to form their own 2-pair teams in advance. If some students drop the course, any remaining pair or team members may be arbitrarily reassigned to other pairs/teams at the discretion of the instructor (but are strongly encouraged to reform pairs/teams on their own). Students will develop and test their project code together with the other member of their programming pair.

Readings

- Readings
  - Slides and Handouts posted on the course web site
  - Documentation provided with Rational RequisitePro
  - Documentation provided with business and application modeling tools (e.g., Popkin Software Architect)
  - SE Textbook: Chapters 3-8 & 18
- Project Frameworks Setup (ongoing)
  - As per references provided on the course Web site
- Individual Assignment
  - See Session 3 Handout: "Assignment #2"
- Team Assignment
  - See Session 2 Handout: "Team Project Specification" (Part 1)

Next Session:

Risk Management in Software Engineering Projects

- Project Planning and Estimation
- Cooperative Roles of Software Engineering and Project Management
- Developing Risk Response Strategies
- Risk Management in Agile Processes
- Agile Project Planning